Using SMT engine to generate Symbolic Automata -Extended version

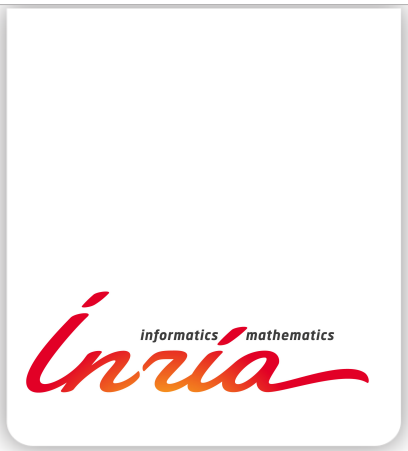
Xudong Qin, Simon Bliudze, Eric Madelaine, Min Zhang

Using SMT engine to generate Symbolic Automata - Extended version

Xudong QIN, Simon BLIUDZE, Eric MADELAINE, Min ZHANG

RESEARCH REPORT
N° 9177
June 2018
Common Project-Team Kairos

ISSN 0249-6399 ISRN INRIA/RR--9177--FR+ENG



Using SMT engine to generate Symbolic Automata - Extended version

Xudong QIN*†, Simon BLIUDZE‡, Eric MADELAINE*, Min ZHANG†

Common Project-Team Kairos

Research Report n° 9177 — June 2018 — 29 pages

Abstract: Open pNets are used to model the behaviour of open systems, both synchronous or asynchronous, expressed in various calculi or languages. They are endowed with a symbolic operational semantics in terms of so-called “Open Automata”. This allows us to check properties of such systems in a compositional manner. We implement an algorithm computing these semantics, building predicates expressing the synchronization conditions between the events of the pNet subsystems. Checking such predicates requires symbolic reasoning over first order logics, but also over application-specific data. We use the Z3 SMT engine to check satisfiability of the predicates, and prune the open automaton of its unsatisfiable transitions. As an industrial oriented use-case, we use so-called "architectures" for BIP systems, that have been used in the framework of an ESA project and to specify the control software of a nanosatellite at the EPFL Space Engineering Center. We use pNets to encode a BIP architecture extended with explicit data, and compute its open automaton semantics. This automaton may be used to prove behavioural properties; we give 2 examples, a safety and a liveness property.

Key-words: Symbolic behavioral semantics, Compositional analysis of software systems, Networks of synchronized automata

---

KAIROS is a common team between the I3S Lab (Univ. of Nice Sophia-Antipolis and CNRS) and INRIA Sophia Antipolis Méditerranée.

This work was partially funded by the Associated Team FM4CPS between Inria and ECNU Shanghai.

* Université Côte d’Azur, INRIA, CNRS, 06902 Sophia Antipolis, France
† Shanghai Key Laboratory for Trustworthy Computing, East China Normal University, Shanghai, China
‡ INRIA Lille-Nord Europe, 40 avenue Halley, 59650 Villeneuve d’Asq, France

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Utilisation d'un Moteur SMT pour générer des Automates Symboliques- Version étendue

Résumé : Les pNets ouverts sont utilisés pour modéliser le comportement des systèmes ouverts, synchrones ou asynchrones, exprimée dans divers calculs ou langages de programmation. Ils sont dotés d'une sémantique opérationnelle symbolique en termes d'«Automata Ouverts». Cela nous permet de vérifier les propriétés de ces systèmes d'une manière compositionnelle. Nous avons implémenté un algorithme calculant ces sémantiques, en construisant des prédicats exprimant les conditions de synchronisation entre les actions des composants du pNet. La vérification de tels prédicats nécessite un raisonnement symbolique sur les logiques de premier ordre, mais également sur des données spécifiques à l'application. Nous utilisons le moteur SMT Z3 pour vérifier la satisfiabilité des prédicats, et ne conserver dans l'automate ouvert que les transitions satisfiables.

Nous illustrons notre approche par un exemple d'inspiration industrielle. Pour cela nous partons d'«architectures» de systèmes BIP, qui ont été utilisés dans le cadre d'un projet de l'Agence Spatiale Européenne pour spécifier le logiciel de contrôle d'un nanosatellite au Centre d'ingénierie spatiale de l'EPFL. Nous utilisons les pNets pour encoder une architecture BIP étendu avec des données explicites, et calculer sa sémantique en termes d'automates ouverts. Cet automate peut être utilisé pour prouver des propriétés comportementales; nous donnons 2 exemples, une propriete de sureté et une de vivacité.

Mots-clés : Sémantique comportementale symbolique, Analyse compositionelle de logiciels, Réseaux d'automates synchronisés

1 Introduction

[1]

In the nineties, several works extended the basic behavioural models based on labelled transition systems to address value-passing or parameterized systems, using various symbolic encodings of the transitions [1, 2, 3, 4]. In [4], H.M. Lin addressed value-passing calculi, for which he developed a symbolic behavioural semantics, and proved algebraic properties. Separately J. Rathke [5] defined another symbolic semantics for a parameterized broadcast calculus, together with strong and weak bisimulation equivalences, and developed a symbolic model-checker based on a tableau method for these processes. Thirty years later, no practical verification approach and no verification platform are using this kind of approaches to provide proof methods for value-passing processes or open process expressions.

Parameterized Networks of Synchronized Automata (pNets) were proposed to give a behavioural specification formalism for distributed systems, synchronous, asynchronous, or heterogeneous. They are used in VerCors [6], a platform for designing and verifying distributed systems, as the intermediate language for various high-level languages. The high-level languages in VerCors formalize each component of the distributed system and their composition. pNets provides the core low-level semantic formalism for VerCors, and is made of a hierarchical composition of (value-passing) automata, called parameterized labelled transition systems (pLTS), where each hierarchical level defines the possible synchronization of the lower levels. Traditionally, pNets have been used to formalize fully defined systems or softwares. But we want also to define and reason about incompletely defined systems, like program skeletons, operators, or open expressions of process calculi. The open pNet model addresses this problem, using "holes" as process parameters, representing *unspecified* subsystems. The pNet model was developed in a series of papers [7, 8] in which many examples have been introduced showing its ability to encode the operators from some other algebras or program skeletons. The operational semantics of an (open) pNet is defined as an Open Automaton in which Open Transitions contain logical predicates expressing the relations between the behaviour of the holes, and the global behaviour of the system. In the previous publication, only a sketch of a procedure allowing to compute these semantics was presented, together with a proof of finiteness of the open automaton, under reasonable hypotheses on the pNet structure.

Implementing these semantics raised several challenges, in order:

- to get a tool that could be applied to pNets representing various languages, in particular various actions algebras, with their specific decision theories,
- to separate clearly the algorithm generating the transitions of the open automaton from combination of all possible (symbolic) behaviours, from the symbolic reasoning part, specifically here using an SMT engine to check the satisfiability of the predicates generated by our algorithm,
- to build a prototype and validate the approach on our basic case-studies, and understand the efficiency of the interaction with the SMT solver.

In the long term, we want to be able to check the equivalence between open systems encoded as pNets. The equivalence between pNets is "FH-bisimulation" [8], a dedicated version of symbolic bisimulation taking the predicate of the open transitions into account when matching such open transitions. We foresee that the interplay with the SMT solver that we use here for satisfiability of open transitions will be similar with what we need when proving (symbolic) equivalence between open transitions.

[1] This Research Report is the extended version of the eponym paper presented at the 18th Int. Workshop on Automated Verification of Critical Systems (AVOCS'18) in Oxford, July 2018.

Contribution In the article we show how:

- We define the open automaton generation algorithm. We implemented a full working prototype, within the VerCors platform. In the process, we improved the semantics rules from [8], and add features in the algorithm to deal with the full model, including management of variables and assignments.
- We implement the interaction between our algorithm and the Z3 SMT solver, for checking satisfiability of the transitions generated by the algorithm.
- We show the interest of this approach on an industry-inspired case-study, namely one architectural pattern extracted (and extended) from the BIP specification of a nanosatellite on-board software.

Related work. Very few attempts were made to develop symbolic bisimulation approaches for the value-passing process algebras and languages—our long-term goals—especially, there is no algorithmic treatment of the symbolic systems developed by interacting with automatic theorem provers. The closest work is the one already mentioned from J. Rathke [5], who developed the symbolic bisimulation for a calculus of broadcasting system (CBS). CBS is similar with classic process calculi such as CCS and CSP, but communicating by broadcasting values, transmitting values without blocking. That makes the definition of the symbolic semantic and bisimulation equivalence different from the classic works.

For other applications, such as the analysis of programming languages, there exist dedicated platforms using external automatic theorem provers (ATP) or automatic tactics from interactive theorem provers (ITP), to perform symbolic reasoning, and for example to discharge some subgoals in the proofs. Tools like Rodin [9, 10, 11] have already integrated several provers, like Z3, as modules for proving the proof obligations generated from a user model. The prover we use, which also happens to be Z3, is developed by Microsoft Research based on the satisfiability modulo theories framework (SMT), is mainly applied in extended static checking, test case generation, and predicate abstraction. In a similar way, there are several ATPs/ITPs we could consider to use for result pruning and bisimulation checking in our algorithm, as an alternative to Z3, such as CVC4 [12], Coq [13], Isabelle [14] or others.

BIP(Behaviour-Interaction-Priority) [15] is a framework for the component-based design of concurrent software and systems. In particular, the BIP tool-set comprises compilers for generating C/C++ code, executable by linking with one of the dedicated engines, which implement the BIP operational semantics [16]. This approach ensures that any property, shown to hold on a given BIP model, will also hold by construction on the generated code. BIP Architectures [17] formalise design patterns, which enforce global properties characterising the coordination among the components of the system. They provide a compositional approach, ensuring correctness by construction during the design of BIP models. In [17], it was shown that application of architectures is compositional w.r.t. safety properties, i.e. when several architectures are applied, each enforcing a safety property, the resulting system satisfies their conjunction.

But the interaction feature in architectures does not handle data-sensitive interaction constraints. Using an encoding of architectures, extended with data-dependant interactions, into open pNets was an interesting alternative to a direct extension to the architecture semantics.

Structure. In section 2 we give a description and a formal definition of the pNet model, as found in previous publications. Then in section 3 we present our use-case, based on a BIP architecture from the nano-satellite case-study. Section 4 recalls briefly the operational semantics of pNet. Section 5 explains in details the algorithm used to compute this semantics, including the interaction with Z3, and shows the full result of the semantic computation on the running example. Finally we conclude and discuss perspectives in Section 6.

2 Background: pNets definition

This section introduces pNets and the notations we will use in this paper. Then it gives the formal definition of pNet structures, together with an operational semantics for open pNets.

pNets are tree-like structures, where the leaves are either *parameterized labelled transition systems (pLTSs)*, expressing the behaviour of basic processes, or *holes*, used as placeholders for unknown processes, of which we only specify their set of possible actions, named *sort*. Nodes of the tree (pNet nodes) are synchronizing artifacts, using a set of *synchronization vectors* that express the possible synchronization between the parameterized actions of a subset of the sub-trees.

Notations. We extensively use indexed structures over some countable indexed sets, which are equivalent to mappings over the countable set. $a_i^{i\in I}$ denotes a family of elements a_i indexed over the set I. When this is not ambiguous, we shall use notations for sets, and typically write "indexed set over I" when formally we should speak of multisets, and write $x \in a_i^{i\in I}$ to mean $\exists i \in I.\, x = a_i$. An empty family is denoted $\emptyset$. We denote classically $\overline{a}$ a family when the indexing set is irrelevant. $\uplus$ is the disjoint union on indexed sets.

Term algebra. Our models rely on a notion of parameterized actions that are symbolic expressions using data types and variables. As we want to encode the low-level behaviour of possibly very different programming languages, we do not want to impose one specific algebra for denoting actions, nor any specific communication mechanism. So we leave unspecified the constructors of the algebra that will allow building expressions and actions. Moreover, we use a generic *action interaction* mechanism, based on unification between two or more action expressions. This will be used in the semantics of synchronization vectors to express various kinds of communication or synchronization mechanisms.

Formally, we assume the existence of a term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$, where Σ is the signature of the data and action constructors, and $\mathcal{V}$ a set of variables. Within $\mathcal{T}_{\Sigma,\mathcal{V}}$, we distinguish a set of data expressions $\mathcal{E}_\mathcal{V}$, including a set of Boolean expressions $\mathcal{B}_\mathcal{V}$ ($\mathcal{B}_\mathcal{V} \subseteq \mathcal{E}_\mathcal{V}$). On top of $\mathcal{E}_\mathcal{V}$ we build the action algebra $\mathcal{A}_\mathcal{V}$, with $\mathcal{A}_\mathcal{V} \subseteq \mathcal{T}_\mathcal{V}, \mathcal{E}_\mathcal{V} \cap \mathcal{A}_\mathcal{V} = \emptyset$; naturally action terms will use data expressions as sub-terms. The function $vars(t)$ identifies the set of variables in a term $t \in \mathcal{T}$.

pNets can encode naturally the notion of input actions as found, e.g. in value-passing CCS [18] or of usual point-to-point message passing calculi, but it also allows for more general mechanisms, like gate negotiation in Lotos [19], or broadcast communications.

Algebra presentations. In practice, the parameterization of the pNet model by some specific action algebra is realized by the definition of a many-sorted "algebra presentation". It will be used to check the well-formedness of a pNet system, and to define the translation of the pNet semantics into the SMT engine input language ([20]).

Definition 2.1 *An* algebra presentation *is a triple* $\mathcal{P} = \langle Sorts, Constrs, Ops\rangle$*, where*

- *Sorts is a set of data* sorts
- *Constrs is a set of* constructor operators*: for each* $Con \in Constrs$*,* $arity(Con) = n \in \mathbb{N}$ *is its arity and '*$Con : (sel_1, sort_1), \ldots, (sel_n, sort_n) \rightarrow sort$*' is its signature with the associated selectors. For each argument, the pair* $(sel_i, sort_i)$ *defines an auxiliary operator of name* sel_i *with signature* $sel_i : sort \rightarrow sort_i$*.*
- *Ops is a set of* auxiliary operators*, with their arity and signature, of the form:* $Op : sort_1, \ldots, sort_n \rightarrow sort$

Table 1: Algebra Presentation: predefined Sorts and Operators

Sort	Constructors	Auxiliary Operators
Bool	`true`, `false`	$\wedge$, $\vee$, $\neg$, $\Longrightarrow$, $=$, $\neq$
Action	`Synchro`, `FUN`	
Int	$0, \{i, -i\}_{i \in \texttt{Nat}}$	$-$(`unary`), $+$, $-$(`binary`), $\times$, $\div$ etc.
Extension for the BIP use-case of Fig. 2		
Action	FUN_Action_Bool, fail, resume, timeout, reset, start, tick, ask	

- *Constrs(sortname) and Sels(sortname) are, respectively, the sets of constructors and selectors of the sort sortname*

Constructors of arity 0 are called constants*, and denoted Consts($\mathcal{P}$).*

Sorts `Bool` and `Int` are predefined with standard operators. Sort `Action` also, with a constructor `Synchro` denoting a synchronized action, i.e. an "internal" action that cannot be further synchronized with the environment. It also comes with an overloaded `FUN` constructor, used to build actions with arguments, that will be instantiated to the required sorts for a given pNet.

The definition of an Algebra Presentation, and a set of variables $\mathcal{V}$ fixes the Term algebra elements $\mathcal{T}_{\Sigma,\mathcal{V}}, \mathcal{B}_{\mathcal{V}}, \mathcal{A}_{\mathcal{V}}$.

2.1 The (open) pNets Core Model

A pLTS is a labelled transition system with variables, which can be manipulated, defined, or accessed inside states, actions, guards, and assignments. Each state has its set of variables called *state variables*, which can only be modified by the assignment in transitions targeting this state. A global state variable of a pLTS is a state variable defined in all states. Note that we make no assumptions on finiteness of the set of states, nor on finite branching of the transition relation.

We first define the set of actions a pLTS can use. Let a range over action labels, op are operators, and x_i range over variable names. Action terms are:

$$\begin{array}{rcll} \alpha \in \mathcal{A} & ::= & a(p_1, \ldots, p_n) & \text{action terms} \\ p_i & ::= & \mathit{Expr} & \text{parameters} \\ \mathit{Expr} & ::= & \mathit{Value} \mid x \mid op(\mathit{Expr}_1, \ldots, \mathit{Expr}_n) & \text{expressions} \end{array}$$

Definition 2.2 (pLTS) *Given a term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$, a pLTS is a tuple $pLTS \triangleq \langle\!\langle S, s_0, \rightarrow \rangle\!\rangle$ where:*

- *S is a set of* states*, with $s_0 \in S$ the* initial state*.*
- *$\rightarrow \subseteq S \times L \times S$ is the* transition relation*, with L the set of labels of the form $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$, where $\alpha \in \mathcal{A}_{\mathcal{V}}$ is a parameterized action, $e_b \in \mathcal{B}_{\mathcal{V}}$ is a guard, and expressions $\mathcal{E}_{\mathcal{V}} \cup \mathcal{A}_P$ are assigned to x_j. If $s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle} s'$ then $vars(e_b) \subseteq vars(s) \cup vars(\alpha)$, and $\forall j \in J.\ vars(e_j) \subseteq vars(s) \wedge x_j \in vars(s')$.*

Now, we define pNet nodes as constructors for hierarchical structures. A pNet node has a set of sub-pNets that can be either pNets or pLTSs, and a set of *holes*, playing the role of process parameters (i.e. unknown in the environment).

A composite pNet consists of a set of sub-pNets, each exposing a set of actions. The relation between actions of a pNet and those of its sub-pNets are given by *synchronization vectors*, which synchronize one or several internal actions, and expose a single resulting global action.

Definition 2.3 (pNets) *A pNet is a hierarchical structure where leaves are pLTSs and holes: $pNet \triangleq pLTS \mid \langle\!\langle pNet_i^{i \in I}, J, SV_k^{k \in K} \rangle\!\rangle$, with I, J, K potentialy infinite, where*

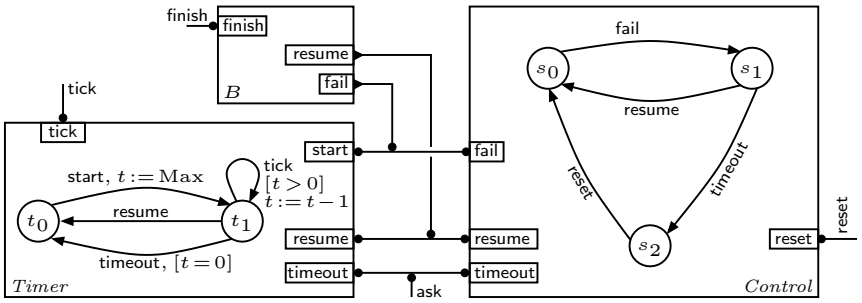


Figure 1: The BIP specification of the Failure Monitor architecture

- $pNet_i^{i \in I}$ *is the family of sub-pNets;*
- J *is a set of indexes, called* holes*.* I *and* J *are* disjoint*:* $I \cap J = \emptyset$*,* $I \cup J \neq \emptyset$
- $SV_k^{k \in K}$ *is a set of synchronisation vectors* ($K \in \mathcal{I}_\mathcal{V}$)*.* $\forall k \in K, SV_k = \alpha_l^{l \in I_k \uplus J_k} \rightarrow \alpha'_k \,|\, g_k$*, where* $\alpha'_k \in \mathcal{A}_\mathcal{V}$*,* $I_k \subseteq I$*,* $J_k \subseteq J$*, and* $vars(\alpha'_k) \subseteq \bigcup_{l \in I_k \uplus J_k} vars(\alpha_l)$*. The global action of a vector* SV_k *is* α'_k*. The Boolean expression* g_k*, such that* $vars(g_k) \subseteq \bigcup_{l \in I_k \uplus J_k} vars(\alpha_l)$*, is a guard associated to the vector.*

In Fig. 2, we show examples of these constructs, with two pLTSs, one hole and one pNet node encoding our running example.

3 Running example

As a running example we use the Failure Monitor architecture from the CubETH nanosatellite on-board software case-study [21] realised using BIP. The architecture-based design process in BIP takes as input a set of components providing basic functionality of the system and a set of temporal properties that must be enforced in the final system. For each property, a corresponding architecture is identified and applied to the model, thereby potentially introducing additional coordinator components and modifying the connectors that define synchronisation patterns among ports of components.

Figure 1 shows a refined version of the Failure Monitor architecture used in [21]. Contrary to standard BIP models, architectures comprise one or several *operand* components, whereof only the set of *ports* is given. Here, the operand component is B and its interface consists of the ports finish, resume and fail. The two *coordinator* components—*Control* and *Timer*—are standard BIP components insofar as they also have their *behaviour* specified by finite automata extended with local data variables. Transitions of these automata are labelled with the ports of the corresponding components, Boolean guards and update functions on local variables. For instance the loop transition $t_1 \xrightarrow{\mathsf{tick},[t>0],t:=t-1} t_1$ in the *Timer* component is labeled by the port tick, it can be fired only when the current value of the local variable t is greater than 0. Upon firing, this transition decrements the value of t by 1. When omitted, the default guard (resp. update function) is the constant predicate *true* (resp. the *skip* operator). The constant Max, in $t_0 \xrightarrow{\mathsf{start},t:=\mathrm{Max}} t_1$, is a parameter of the architecture.

Connectors are hierarchical, tree-like structures with component ports at the leaves. They define sets of *interactions*, based on the attributes of the connected ports [22], which may be either *trigger* (triangles in Fig. 1) or *synchron* (bullets in Fig. 1). If all sub-connectors of a connector are synchrons, then an interaction may be executed by the connector only if each subconnector can contribute. If at least one of the sub-connectors is a trigger, then any interaction consisting of contributions of any set of sub-connectors, *involving at least one of the triggers*, can be executed.

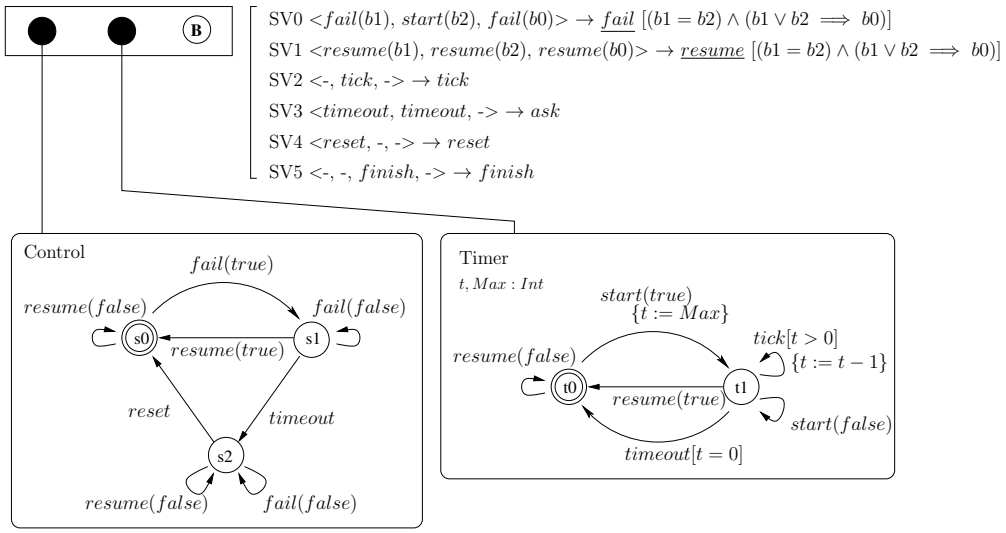


Figure 2: pNet encoding of the Failure Monitor architecture

For instance, the two ports *Timer*.start and *Control*.fail are always synchronised, since they belong to the same binary sub-connector, where they are both synchrons. In particular, this means that whenever the transition $s_0 \xrightarrow{\mathsf{fail}} s_1$ is fired, so is the transition $t_0 \xrightarrow{\mathsf{start},t:=\mathrm{Max}} t_1$, initialising the timer. The binary connector *Timer*.start ●—● *Control*.fail is a sub-connector of a hierarchical connector, where the port B.fail is a trigger. Thus, the above interaction can only happen together with B.fail, forming a ternary interaction. On the contrary, being a trigger, the port B.fail can fire alone, forming a singleton interaction. The composition semantics of BIP systems consists in firing exactly one interaction, enabled through at least one of the top-level connectors, at each execution round.

Finally, *priorities*—defined by a strict partial order on the set of possible interactions—narrow the choice among the enabled interactions at any given round. The default priority is the so-called *maximal progress*, whereby among any two interactions $a \subset b$ (as sets of ports), b has higher priority than a. For example, the port B.fail will never fire alone in a global state, where both *Timer*.start and *Control*.fail are enabled.

Application of the Failure Monitor architecture ensures that, whenever a failure is registered in the operand component, the system will be reset, unless a resumption is registered within Max time units (more details in Sect. 5.4).

Figure 2 shows a pNet encoding of the above Failure Monitor architecture. This encoding is structural: each coordinator component is encoded as a pLTS, the operand component—as a hole; connectors of the BIP model are encoded as synchronisation vectors. Each connector that does not involve triggers is trivially encoded by a synchronisation vector comprising the same ports. In order to encode the semantics of the connectors involving triggers, we 1) in the pLTS encoding the coordinator components, add loop transitions to ensure that all ports involved in such connectors are enabled in all states, 2) associate a Boolean value to each of these ports: the original transitions carry the value *true* (e.g. $s_0 \xrightarrow{\mathsf{fail}(true)} s_1$), the added loops carry the value *false* (e.g. $s_2 \xrightarrow{\mathsf{fail}(false)} s_2$), 3) add to the corresponding synchronisation vectors the Boolean predicate encoding the connector structure. For example, $SV0$ encodes the connector discussed above: the predicate $(b1 = b2) \wedge (b1 \vee b2 \Rightarrow b0)$ means that the "true" transitions *Control*.fail and *Timer*.start can only fire together $(b1 = b2)$ and whenever one of them fires, B.fail must fire also $(b1 \vee b2 \Rightarrow b0)$. This encoding can be systematically obtained for any hierarchical BIP

connector [23]. Although, for the sake of brevity, we omit priorities from the encoding, this can be easily achieved, by introducing additional Boolean variables for relevant ports [16].

4 Operational Semantics for Open pNets

The semantics of open pNets will be defined as an open automaton, that is an automaton where each transition composes transitions of several LTSs with the actions of some holes; the transition occurs if some predicates hold, and can involve a set of state modifications. Each state of an open automaton has a set of *state variables* that can be assigned by incoming transitions. Strictly speaking, the LTSs at the leaves of the open automaton are a restricted form of pLTSs, where labels are parametrised actions, but include no guard nor assignments.

Definition 4.1 (Open transitions) *An* open transition *OT over a set $\langle S_i, s_{0i}, \rightarrow_i \rangle^{i \in I}$ of LTSs, a set J of holes, and a set of states $\mathcal{S}$ is a structure of the form:*

$$\frac{\{s_i \xrightarrow{a_i}_i s'_i\}^{i \in I}, \{\xrightarrow{b_j}_j\}^{j \in J}, Pred, Post}{s \xrightarrow{\alpha} s'},$$

where $s, s' \in \mathcal{S}$, the a_i, b_j, α are action expressions, $s_i \xrightarrow{a_i}_i s'_i$ is a transition of the LTS $\langle S_i, s_{0i}, \rightarrow_i \rangle$, b_j is an action of the hole j, and α is the resulting action of OT. $Pred$ is a predicate over the variables of the terms, labels, and states s_i, b_j, s, α. $Post$ is a set of equations that hold after the open transition*, represented as a substitution $\{x_k \leftarrow e_k\}^{k \in K}$ where x_k are variables of s' and s'_i, whereas e_k are expressions over the other variables of the open transition.*

Definition 4.2 (Open automaton) *An* open automaton *is a structure $A = \langle LTS_i^{i \in I}, J, \mathcal{S}, s_0, \mathcal{T} \rangle$ where:*

- *I and J are sets of indices,*
- *$LTS_i^{i \in I}$ is a family of LTSs,*
- *$\mathcal{S}$ is a set of states and $s_0 \in \mathcal{S}$ the initial state,*
- *$\mathcal{T}$ is a set of open transitions and, for each $t \in \mathcal{T}$, there exist I', J' with $I' \subseteq I$, $J' \subseteq J$, such that t is an open transition over $LTS_i^{i \in I'}$, J', and $\mathcal{S}$.*

The *states* and the shape of *predicates* in the transitions of an open automaton representing the semantics of a pNet have the following specific structure.

States of open pNets. A state of an open pNet is a tuple of the states of its leaves (in which we denote tuples in structured states as $\triangleleft \ldots \triangleright$). For any pNet p, let $\overline{Leaves} = \langle\!\langle S_i, s_{i0}, \rightarrow_i \rangle\!\rangle^{i \in L}$ be the set of pLTS at its leaves, then $States(p) = \{\triangleleft s_i^{i \in L} \triangleright \mid \forall i \in L. s_i \in S_i\}$. A pLTS being its own single leave: $States(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle) = \{\triangleleft s \triangleright \mid s \in S\}$. The initial state is defined as: $InitState(p) = \triangleleft {s_{i0}}^{i \in L} \triangleright$.

Predicates. Let $\langle\!\langle \overline{pNet}, J, SV_k^{k \in K} \rangle\!\rangle$ be a pNet. Consider a synchronization vector SV_k, for $k \in K$. We build a predicate *MkPred* relating the actions of the involved sub-pNets and the resulting actions. This predicate verifies:

$$MkPred(SV_k, a_i^{i \in I}, b_j^{j \in J}, v) \iff \exists (a'_i)^{i \in I}, (b'_j)^{j \in J}, v'.$$
$$SV_k = (a'_i)^{i \in I}, (b'_j)^{j \in J} \rightarrow v' \wedge \forall i \in I.\, a_i = a'_i \wedge \forall j \in J.\, b_j = b'_j \wedge v = v'$$

Example 4.3 (An open transition) *This transition is generated by application of the vector* `SV0`*, synchronizing the initial actions of pLTSs* `Control` *and* `Timer` *(see Fig. 2), with an action of the hole* `B` *equal to* `fail`*. The local variable* `t` *of the* `Timer` *is assigned to its initial value* `Max`*. A full output of the use-case is provided in appendice F.*

$$\frac{\{s0 \xrightarrow{fail(true)} s1, t0 \xrightarrow{start(true)} t1\}\{\xrightarrow{hb}\}, hb = fail(true) \wedge v = \underline{fail}, \{t := Max\}}{\triangleleft s0, t0 \triangleright \xrightarrow{v} \triangleleft s1, t1 \triangleright}$$

Structural Semantic Rules: The semantics of pNet in term of open automata has been defined in [8], in the form of 2 structural rules, one for pLTSs, one for pNet nodes. For convenience we also include these rules in App. A.

In [8] we also proved that a pNet with finitely many pLTSs, holes, and synchronisation vectors, and with finite pLTSs, has a finite semantics. Remark that all the elements of such pNets and open automata being symbolic, they can represent many classes of unbounded systems.

5 Generation of Open Automata

In this section we describe the algorithm implementing the pNet semantics, the interaction with the Z3 SMT solver, and we show the result on our example.

Algorithm 1 starts with an open pNet, and builds its set of open transitions. Its main loop is a classical residual algorithm: starting from the initial global state, it picks a state in an unexplored set, and computes all possible OTs, adding their target states in the unexplored set, until this set is empty. As the set of global states is finite (under the conditions above), this terminates.

Algorithm 1 Open Automaton Generation

Input: A pNet P (cannot be a hole)

1: Initialize sets $U = \{InitState(P)\}$ and $E = \emptyset$, for unexplored and explored global states, respectively; $L = \emptyset$ for the resulting OTs;
2: **while** $!isEmpty(U)$ **do**
3: Choose S in U; remove S from U, add S to E;
4: $OTs = MakeTransitions(P, S)$;
5: **for** each $OT \in OTs$ **do** Check satisfiability of OT using the SMT solver;
6: **if** $SAT(OT)$ **then**
7: {Add OT to L;
8: Let S' be the target global state of OT
9: **if** $(S' \notin U \cup E)$ **then** Add S' into U; }
10: **end for**
11: **end while**
12: **return** $OA = (InitState(P), L)$;

The inside loop (*MakeTransitions* method) applies recursively the semantic rules following the structure of the pNet. When applied to a pLTSs at the leaves, we simply take the pLTS transitions of the corresponding local state and use the semantic rule (see TR1 in App. A) to build the OT.[2] When applied to a pNet node we use two methods, combining and matching, to

[2]We omit detailed presentation of this case for the sake of brevity.

generate the open transitions in a hierarchical manner, as shown in Alg. 2. This method directly manages the holes of the node, so *MakeTransitions* is never called on a hole.

At the root of the pNet, the predicate of each OT is translated into SMTlib assertions, and checked for satisfiability. The final open automaton includes all satisfiable OTs, and the set of reachable global states.

Algorithm 2 MakeTransitions() *for a pNet node*

Input: a pNet node P with subnets $\overline{sn}$ and holes $\overline{hole}$; a global state S.

1: Initialize empty list l and set L for sub-transitions and transitions, respectively;
2: **for** each *Subnet* in $\overline{sn}$ **do** \\ Recursively apply the semantic rules on the subnets
3: Store $MakeTransitions(Subnet, S)$ in l;
4: **end for**
5: $\overline{comb} = Combining(l)$;
6: **for** each $sv \in SV$ and each $comb \in \overline{comb}$ **do**
7: $ot = Matching(sv, \overline{comb}, \overline{hole})$;
8: **if** (ot is defined) **then** Store ot in L; \\ if *Matching*() succeeds
9: **end for**
10: **return** L;

Combining. The combining method enumerates all the possible behaviours of the subnets as all the possible combinations of their open transitions. Assume that there is a collection of n subnets. We denote $\overline{ot}_i$ the set of open transitions of the i-th subnet (obtained in line 3 of the algorithm); "$-$" means that the subnet is not involved. The combination $\overline{comb}$, a set of n-tuples, is the cartesian product: $\overline{comb} = (\{-\} \cup \overline{ot}_1) \times (\{-\} \cup \overline{ot}_2) \times \cdots \times (\{-\} \cup \overline{ot}_n)$.

Matching. The *Matching* method builds the OTs of a pNet node from those of its subnets (see rule Tr2 in App. A). For each synchronisation vector and each possible combination of behaviours of the subnets, as generated by the *Combining* method, it builds the corresponding open transition. Here, we only detail the construction of the predicate. From a synchronization vector $sv = \left((a'_i)^{i\in I}(b'_j)^{j\in J} \to v'\right) \in SV$ and its guard G_k; a tuple of open transitions $C = (ot_i)^{i\in[1,n]} \in \overline{comb}$, such that, for each $i \in [1,n]$, either $ot_i = -$, or the result action of ot_i is a_i; the hole behaviours $Hole = (b_j)^{j\in J}$; and a fresh variable v, representing the result action of the OT under construction, we build the predicate:

$$MkPred(sv, C, \overline{hole}) = (\forall i \in I, a_i = a'_i) \wedge (\forall j \in J, b_j = b'_j) \wedge (v = v') \wedge G_k$$

Filtering. While matching a vector with a combination tuple, *Matching* tries to filter out some incompatibilities; there may be several reasons why the matching would fail:

- if some subnet is marked as inactive in the vector, and the chosen combination has an active behaviour at this position,
- if some subnet action expression in the vector does not match (by pattern-matching) the corresponding action expression in C,
- or if the whole set of active subnet actions in the vector cannot be matched (by unification) with the corresponding action expressions on the tuple C.

Even when unification succeeds, it is still possible that the resulting predicate would be unsatisfiable, because of some incompatibility involving the guards. In our algorithm, we choose to apply only the simplest filter inside the *Matching* method (before applying the predicate and

$$ot = \frac{\{s0 \xrightarrow{fail(true)} s0, t0 \xrightarrow{resume(false)} t0\},\ \{\xrightarrow{hb}\},\ fail(true) = fail(b_1) \wedge resume(false) = start(b_2) \wedge hb = fail(b_0) \wedge v = \underline{fail} \wedge b_1 = b_2 \wedge (b_1 \vee b_2) \Rightarrow b_0,\ \{\}}{\lhd s0, t0 \rhd \xrightarrow{v} \lhd s0, t0 \rhd}$$

Figure 3: One of the unsatisfiable open transitions in the Failure Monitor pNet

OT construction). Matching and unification will be checked later, together with the guards collected from synchronisation vectors and from pLTS transitions, using the satisfiability check in the SMT engine.

5.1 Management of state variable assignments

In a pLTS, there may be several incoming transitions that assign potentially different values to a state variable. To handle such cases, the algorithm manages, for each pLTS state, a list of expressions collected from the assignments of each state variable. For a global state in the open automaton, the set of state variables (which may be used in a transition) is the disjoint union of sets of state variables of the individual pLTS states constituting this global state.

5.2 Pruning the unsatisfiable results

Our matching/filtering strategy builds some transitions where the predicates express incompatible constraints. Even if having an unsatisfiable (symbolic) transition would not be incorrect, we choose to minimize the open automaton (i.e. its number of transitions and states), by checking the predicates for satisfiability. In Fig. 3, we show an unsatisfiable open transition from the open automaton of our running example. It shows the case where the failure controller performs a "fail" action, while the timer executes a "resume". The chosen synchronization vector (SV0 from Fig. 2) does not match with these actions, since it expects *Timer*.start. This mismatch is materialised by the predicate fragment "$resume(false) = start(b_2)$".

Checking satisfiability requires some symbolic computation on the action expressions and the predicates, which may depend on the specific theory of the action algebra datatypes. The "Modulo Theory" part of SMT solvers is important here, so that the solver can use specific properties of each action algebra.

5.3 Translation to SMTlib

We check the satisfiability of each open transitions using the SMT solver Z3. In this section, we describe the translation of the algebra presentation, of assignments, and of the predicates.

Our implementation submits satisfiability requests to Z3 using its JAVA API. Here, for readability, we show the Z3 code using its SMT-LIB input language. Note that in the previous sections, the OTs were displayed in a simplified, human readable form. The input and output of our tool, and also the generated SMTlib fragments, are slightly more difficult to read, in particular because of structured names for the fresh variables generated by the algorithm, allowing tracability of the result (see appendix B.1).

Figure 4 shows the translation of the transition of Fig. 3 in the SMTlib syntax. It contains the declaration of the BIP action algebra sorts and constructors, then the declaration of variables, and finally the predicate to be checked, encoded as a set of assertions. Here we display also the diagnosis ("sat" or "unsat") generated by Z3;

```
(declare-datatypes () ((Action (fail)(resume)(timeout)(reset)(start)(tick)(ask)
                (FUN_Action_Bool (fst Action)(snd Bool))(Synchro (action Action)))))
(declare-const |b1:sva_SV0:1:1| Bool)
(declare-const |b2:sva_SV0:1:1| Bool)
(declare-const |b0:sva_SV0:1:1| Bool)
(declare-const |:hb_B:13:1| Action)
(declare-const |:ra:1:1| Action)
(assert (= (FUN_Action_Bool fail true) (FUN_Action_Bool fail |b1:sva_SV0:1:1|)))
(assert (= (FUN_Action_Bool resume false) (FUN_Action_Bool start |b2:sva_SV0:1:1|)))
(assert (= |:hb_B:13:1| (FUN_Action_Bool fail |b0:sva_SV0:1:1|)))
(assert (= |b1:sva_SV0:1:1| |b2:sva_SV0:1:1|))
(assert (or (not (or |b1:sva_SV0:1:1| |b2:sva_SV0:1:1|)) |b0:sva_SV0:1:1|))
(assert (= (Synchro fail) |:ra:1:1|))
(check-sat)
```

```
unsat
```

Figure 4: The input of the Z3 solver in SMT-LIB language and the output result

Production of the SMT-lib code To build the input submitted to Z3 for each OT, we translate the algebra presentation, the predicates and the variable assignments into Z3 (Java-API) calls.

Translation of action algebra presentation. In Appendix D and E, we define the condition of well-formedness of an algebra presentation, and its translation into SMTlib declarations (`declare-datatypes` and `declare-fun`). It ensures that the generated code is correct, and will not raise runtime errors in the SMT engine. Note that the `declare-datatypes` command comprises both the action constructors from table 1 and also the constant action names from our example. Additionally, we will include axiomatisation of any required functions and predicates of the algebra data-types.

Translation of open transitions. In App. E we formally define all steps of the translation of each open transition, including:

- collect all variables in the transition, and declare them (using `declare-const`)
- check well-formedness and correct typing of expressions,
- translate the predicate into a conjunction of assertions,
- if present in the source state, translate the state-variable assignments into a disjunctive assertion.

This translation ensures that no runtime error will occur in the SMT engine.

Figure 4 shows the decomposition of the predicate into a set of asserts, each encoding an elementary equality, inequality, or a guard. The result (sat or unsat) of the final `check-sat` command in the translation is then decoded.

5.4 Result for the running example

For this example, the tool builds 184 open transitions, whereof 173 are detected unsatisfiable by Z3. The resulting open automaton, with 3 reachable global states (out of the possible 6) and 11 open transitions, is shown in Fig. 5.

To improve the readability of this figure, we used the following conventions: we omit the transitions of the two pLTS, and the set of "working" holes; and we directly write the resulting action as first element of each OT, rather than including it as an equality inside the predicate. Notice, however that the loop transitions *resume* in states $\triangleleft s0, t0 \triangleright$ and $\triangleleft s2, t0 \triangleright$, corresponding, respectively, to open transitions ot_2 and ot_9 in Fig. 6 of Appendix F, involve `resume(false)`

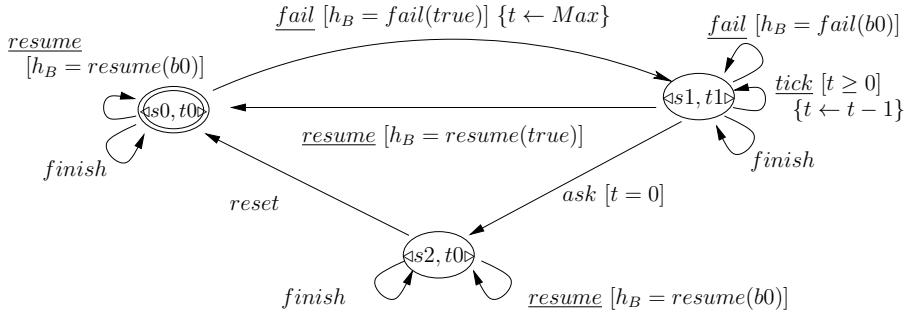


Figure 5: Open Automaton for the Failure Monitor architecture

from the two pLTS, i.e. only the hole, corresponding to the operand component of the Failure Monitor architecture, executes resume.

Failure Monitor enforces 1) the safety property *"the system reset never happens, unless asked for by a timeout following a failure"*, formalised in CTL by

$$\varphi \wedge \mathtt{AG}(\mathsf{reset} \rightarrow \varphi), \quad \text{where } \varphi = \mathtt{A}[\neg \mathsf{reset}\ \mathtt{W}\ \mathsf{ask}] \wedge \mathtt{A}[\neg \mathsf{ask}\ \mathtt{W}\ \mathsf{fail}],$$

(W being the *weak until* operator) and 2) the liveness property *"a reset will be fired when asked for by a timeout"*: AG(ask → AF reset). The satisfaction of the safety property could be established by applying symbolic model checking techniques. However, in this example, it is obvious by inspection of the open automaton. The satisfaction of the liveness property relies on the above observation that in the state $\triangleleft s2, t0 \triangleright$ only the reset transition involves *Control*. Therefore, under reasonable scheduling assumptions, reset will always be fired.

6 Conclusion and Discussion

The formal definitions and properties of the open pNets model were published in [8]. In this new work we describe an implementation of the model and its semantics construction, including its interaction with the Z3 SMT engine. The implementation has two parts: the first is a finitary algorithm that builds all possible combinations of synchronisations through the pNet hierarchical structure. The result is a so-called *open-automaton*, which transitions contains predicates relating the actions of the pNet holes and controllers. Some of the open transitions obtained at this step, may contain predicates which do not represent any possible concrete instantiations. In the second part of the tool we use the SMT solver Z3 for checking the satisfiability of the predicate in each open transition. To this end, we encode into Z3 the representations of the action algebra and the predicates before submitting them to the Z3 solver. In this paper, we used a running example, based on a BIP architecture from an earlier nanosatellite case study [21]. This example shows that open-automata-based semantics can be instrumental in verifying the properties enforced by the architectures through an encoding into open pNets. This encoding—which we intend to formalise and prove correct in a separate paper—also opens the way for an extension of BIP architectures with the transfer of data among variables of different components. Indeed, such data transfer can be easily encoded using the predicates associated to synchronisation vectors in open pNets. The encoding of open transitions into SMTlib and the availability of theories can guide the definition of such an extension. Our case-studies show that our encoding successfully identifies the unsatisfiable open transitions and that the resulting automata correctly reflect the expected movements of the encoded process expressions.

Naturally, our next goals after the generation of the open automata will be to model-check logical properties, and to check equivalence of pNets. While model-checking open automata

seems easy to define, equivalence checking is more challenging. In [8], we have already found the FH-bisimulation, to be a suitable definition. But weak equivalences, or refinements, will definitely be useful when comparing different pNets with different structure. For bisimulation, we foresee that SMT methods will be the basis for comparison of open transitions.

Scaling up. One important motivation of this work is to attack the complexity of verification of realistic systems by a compositional and parametric approach. Still one may wonder if the price for analysing our symbolic transitions will not make the approach too expensive in term of computing time. We tried a slighly bigger example, assembling 2 Failure controllers. In appendix F.1, we show that Z3 can check the satisfiablility of a 90K open transitions in a couple of minutes.

References

[1] De Simone, R.: Higher-level synchronising devices in MEIJE-SCCS. Theoretical Computer Science **37** (1985) 245–267

[2] Larsen, K.G.: A context dependent equivalence between processes. Theoretical Computer Science **49** (1987) 184–215

[3] Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science **138**(2) (1995) 353–389

[4] Lin, H.: Symbolic transition graph with assignment. In Montanari, U., Sassone, V., eds.: Concur'96. Volume 1119 of LNCS., Springer, Heidelberg (1996) 50–65

[5] Hennessy, M., Rathke, J.: Bisimulations for a calculus of broadcasting systems. Theoretical Computer Science **200**(1-2) (1998) 225–260

[6] Henrio, L., Kulankhina, O., Liu, D., Madelaine, E.: Verifying the correct composition of distributed components: Formalisation and Tool. In: FOCLASA. Number 175 in EPTCS, Rome, Italy (2014)

[7] Henrio, L., Madelaine, E., Zhang, M.: pNets: an Expressive Model for Parameterised Networks of Processes. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'15)

[8] Henrio, L., Madelaine, E., Zhang, M.: A Theory for the Composition of Concurrent Processes. In: Formal Techniques for Distributed Objects, Components, and Systems (FORTE). Volume LNCS-9688., Heraklion, Greece (2016)

[9] Déharbe, D.: Integration of smt-solvers in b and event-b development environments. Science of Computer Programming **78**(3) (2013) 310–326

[10] Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating smt solvers in rodin. Science of Computer Programming **94** (2014) 130–143

[11] Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundamenta Informaticae **77**(1-2) (2007)

[12] Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer aided verification, Springer (2011)

[13] Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of sat/smt solvers to coq through proof witnesses. In: International Conference on Certified Programs and Proofs, Springer (2011) 135–150

[14] Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in isabelle/hol. In: International Symposium on Frontiers of Combining Systems, Springer (2011) 12–27

[15] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software **28**(3) (2011) 41–48

[16] Baranov, E., Bliudze, S.: Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP. Science of Computer Programming **109**(0) (2015) 2–35

[17] Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. Formal Aspects of Computing **18**(2) (2016) 207–231

[18] Milner, R.: Communication and Concurrency. Int. Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey (1989) SU Fisher Research 511/24.

[19] ISO: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organisation for Standardization, Geneva, Switzerland (1989)

[20] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017) Available at `www.SMT-LIB.org`.

[21] Mavridou, A., Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis, J.: Architecture-based design: A satellite on-board software case study. In: 13th Int. Conf. on Formal Aspects of Component Software (FACS 2016). (2016)

[22] Bliudze, S., Sifakis, J.: The algebra of connectors—Structuring interaction in BIP. IEEE Transactions on Computers **57**(10) (2008) 1315–1330

[23] Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design **36**(2) (2010) 167–194

These appendices include details that were kept out of the original paper by lack of space. More precisely, they contain:

A pNet semantic rules

Note: For convenience, we have included here the semantic rules for pNets, that were already published in [8]. These rules are referenced from section 5, 10, and implemented by the algorithms 1 & 2. Compared to the rules in [8], they were slightly extended to take into account guards in the synchronisation vectors.

We build the semantics of an open pNet as an open automaton where LTSs are the pLTSs at the pNet leaves, and the states are structured as in the previous section. To build an open transition one first projects the global state into states of the leaves, then applies pLTS transitions on these states, and compose them with actions of holes using synchronisation vectors.

The semantics regularly instantiates *fresh* variables, and uses a *clone* operator that clones a term replacing each variable with a fresh one. The variables in each synchronization vector are considered local: for a given pNet expression, we must have fresh local variables for each occurrence of a vector (= each time we instantiate rule Tr2). Similarly the state variables of each copy of a given pLTS in the system, must be distinct, and those created for each application of Tr2 have to be fresh and all distinct.

Definition A.1 (Operational semantics of open pNets) *The semantics of a pNet p is an open automaton $A =< Leaves(p), J, \mathcal{S}, s_0, \mathcal{T} >$ where:*

- *J is the indices of the holes: $Holes(p) = H_j^{j\in J}$.*
- *$\overline{\mathcal{S}} = States(p)$ and $s_0 = InitState(p)$*
- *$\mathcal{T}$ is the smallest set of open transitions satisfying the rules below:*

*The rule (***Tr1***) for a pLTS p checks that the guard is verified and transforms assignments into post-conditions:*

$$\textbf{Tr1:}\quad \frac{s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle} s' \in \rightarrow \qquad \texttt{fresh}(v) \qquad Pred = \ e_b \wedge (v = \alpha)}{p = \langle\!\langle S, s_0, \rightarrow \rangle\!\rangle \models \frac{\{s \xrightarrow{\alpha}_p s'\}, \emptyset, Pred, \{x_j \leftarrow e_j\}^{j\in J}}{\lhd s \rhd \xrightarrow{v} \lhd s' \rhd}}$$

*The second rule (***Tr2***) deals with pNet nodes: for each possible synchronization vector applicable to the rule subject, the premisses include one* open transition *for each sub-pNet involved, one possible* action *for each Hole involved, and the predicate relating these with the resulting action of the vector. A key to understand this rule is that the open transitions are expressed in terms of the leaves and holes of the pNet structure, i.e. a flatten view of the pNet: e.g. L is the index set of the Leaves, L_k the index set of the leaves of one subnet, so all L_k are disjoint subsets of L.*

Tr2:

$$\frac{\begin{array}{c} k\in K \qquad SV{=}clone(SV_k){=}\alpha_m^{m\in I_k\uplus J_k}\to\alpha'_k, G_k \\ Leaves(p){=}pLTS_l^{l\in L} \qquad \forall m\in I_k.pNet_m \models \dfrac{\{s_i \xrightarrow{a_i}_i s'_i\}^{i\in I'_m}, \{\xrightarrow{b_j}_j\}^{j\in J'_m}, Pred_m, Post_m}{\lhd s_i^{i\in L_m}\rhd \xrightarrow{v_m} \lhd s'_i{}^{i\in L_m}\rhd} \\ I' = \biguplus_{m\in I_k} I'_m \qquad J' = \biguplus_{m\in I_k} J'_m \uplus J_k \qquad Pred = \bigwedge_{m\in I_k} Pred_m \wedge MkPred(SV, v_m^{m\in I_k}, b_j^{j\in J_k}, v) \\ \forall j{\in}J_k.\texttt{fresh}(b_j) \qquad \texttt{fresh}(v) \qquad \forall i\in L\backslash I'.\, s'_i = s_i \end{array}}{p = \langle\!\langle pNet_i^{i\in I}, S_j^{j\in J}, SV_k^{k\in K}\rangle\!\rangle \models \dfrac{\{s_i \xrightarrow{a_i}_i s'_i\}^{i\in I'}, \{\xrightarrow{b_j}_j\}^{j\in J'}, Pred, \uplus_{m\in I_k} Post_m}{\lhd s_i^{i\in L}\rhd \xrightarrow{v} \lhd s'_i{}^{i\in L}\rhd}}$$

In rule TR2, the generated predicate is composed of the conjunction of the predicates of the subnets' OTs, with the additional part encoding the application of the chosen synchronisation vector. In [8] this last part is defined as:

$$MkPred(SV_k, a_i^{i\in I}, b_j^{j\in J}, v) \Leftrightarrow \begin{array}{c} \exists (a'_i)^{i\in I}, (b'_j)^{j\in J}, v'.\, SV_k = (a'_i)^{i\in I}, (b'_j)^{j\in J} \to v' \\ \wedge \forall i\in I.\, a_i = a'_i \wedge \forall j\in J.\, b_j = b'_j \wedge v = v' \end{array}$$

Within our algorithm, these subsets have been computed by the *Combining* method and passed as arguments to the *Matching* method (see section 5)

Termination. To have some practical interest, it is important to know when this algorithm terminates. The following theorem shows that an open-pNet with finite synchronization sets, finitely many leaves and holes, and each pLTS at leaves having a finite number of states and (symbolic) transitions, has a finite automaton. The proof can be found in [8].

Theorem A.2 (Finiteness of open-automata.) *Given an open pNet $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k\in K}\rangle\!\rangle$ with leaves $pLTS_i^{i\in L}$ and holes $Hole_j^{j\in J}$, if the sets L and J are finite, if the synchronization vectors of all pNets included in $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k\in K}\rangle\!\rangle$ are finite, and if $\forall i\in L.\, finite(states(pLTS_i))$ and $pLTS_i$ has a finite number of state variables, then Algorithm 1 terminates and produces an open automaton $\mathcal{T}$ with finitely many states and transitions.*

Given an open pNet $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k\in K}\rangle\!\rangle$ with leaves $pLTS_i^{i\in L}$ and holes $Hole_j^{j\in J}$, build its semantics as in algorithm 1.

We have:

$$finite(L) \wedge finite(J) \wedge \forall i\in L finite(\{s_i\}) \to finite(\mathcal{T})$$

B Variable management

In this section we give more details on two facets of variable management in the tool: the generation of variables names, and the management of state variable assignments. These were described shortly in Section 5.1,5.3.

B.1 Fresh variables

The variables in each synchronisation vector are considered local: for a given pNet expression, we must have fresh local variables for each occurrence of a vector (= each time we instantiate rule Tr2). Similarly the state variables of each copy of a given pLTS in the system, must be

distinct, and those created for each application of Tr2 have to be fresh and all distinct. This will be implemented within the open-automaton generation algorithm, e.g. using name generation using a global counter as a suffix.

Application of the semantic rules require generating a lot of fresh names, for different kind of variables. We could use a global name generator to guarantee unicity, but at the same time, we also hope them still readable. Here we rename the variables with a regular format using the fresh function. More precisely, the fresh function generates a new name adding a suffix after the original name. The suffix contains three parts combined by a colon.

Definition B.1 (Fresh variable) *The format of* fresh variable *(renaming) is defined as:*

prefix : tree index : counter.

- *prefix identifies the kind of the variable. Current kinds are: sva_ <SV> (action variable in vector SV), ra (result action), hb_ <B> (behaviour of hole B).*
- *tree index is the index of the node containing the variable in the tree-like structure of the pNet.*
- *counter is the current value of the corresponding counter.*

For example, in the running example, in the raw output listing in Appendix F, in the first OT, you find variables:

- `:hb_B:13:1` that is the behaviour of hole "B"
- `b1:sva_SV0:1:1` variable "b1" from vector "SV0"
- `:ra:1:1` result action of the current OT

B.2 Management of state variables

In a pLTS, there may be several incoming transitions that assign potentially different values to a state variable. To handle such cases, the algorithm manages for each pLTS state, as a list of expressions collected from the assignments of each state variable v (as registered in the incoming OTs). For a global state in the open automaton, getting the set of state variables (which may be used in a transition) will simply be the union of state variables of the individual pLTS states constituting this global state, as variables of different pLTS are distinct.

We define:

- $SVars(gs)$ the state variables of all states in the global state gs
- $Assigns(svar)$ the set of possible assignments (= expressions) of variable $svar$.

These will be used when translating the predicates into SMTlib language.

C Algebra presentations

In order to submit satisfiability problems to Z3 (for the predicates in open transitions), we need to generate SMTlib programs, from the pNet Algebra presentation and predicates. This is the first part of the translation introduced in section 5.3. More precisely, we need to translate to SMTlib:

- the presentation of the action algebra (sorts and operators) that is defined for a given language (process calculus, or high level programming language),
- for a given pNet, the set of local constants (actions or auxiliary data) that are used in the pLTSs,
- for each open transition: the declaration of variables, and the predicate (including action expressions).

Table 2: Algebra Presentation: predefined Sorts and Operators

Sort	Constructors	Auxiliary Operators
Bool	`true`, `false`	$\wedge$, $\vee$, $\neg$, $\Longrightarrow$
Action	`FUN`	
Int	$0, \{i, -i\}_{i \in \texttt{Nat}}$	$-$`(unary)`, $+$, $-$`(binary)`, $\times$, $\div$, `etc.`
for any sort		$=$, $\neq$

Table 3: Additional operator for the BIP algebra

Sort	Constructors	Auxiliary Operators
Action	`FUN_Action_Bool`	

Table 4: Additional sorts and operators for the CCS algebra

Sort	Constructors	Auxiliary Operators
Channel, Data	$\emptyset$	$\emptyset$
Action	`Emit(c,v)`, `Receive(c,v)`, `Tau`	

During this translation, in order to guarantee that the generated code will cause no runtime errors during parsing and execution, we need to ensure that all objects used in the SMTlib code are properly declared, and that they are correctly typed.

Note that in principle, an action algebra corresponds to a given high-level language (e.g. a process algebra), and that the algebra presentation will be defined once and for all in the framework of the pNet semantics of each specific language.

C.1 Definition of an algebra presentation

We have a minimal, predefined algebra presentation for all pNets, including three basic sorts *Bool*, *Action* and *Int* and their operators. Table 2 defines these elements.

In addition, and for convenience, we provide one generic construct for parameterized actions named FUN, which can accept any number of arguments of any sort. The result type, though, can only be *Action*, in order to keep the type-checking simple (see next section).

For a given language, or for a given use-case, the designer can declare more sorts and operators, using our pNet API. As an example, a (value-passing) CCS action algebra, where we assume a single auxiliary value domain "Data", can be defined as:

Example C.1

- $Sorts_{CCS} = \{Action, Channel, Data, Int, Bool\}$
- $Constrs_{CCS} =$
  $\texttt{Emit} : 2, \{(Chan\_E : Channel), (Value\_E : Data)\} -> Action$
  $\texttt{Receive} : 2, \{(Chan\_R : Channel), (Value\_R : Data)\} -> Action$
  $\texttt{Tau} : 0, \{\} -> Action$
  ... and all predefined operators

Then in pNet API these operators or constants can be declared as:

```
AlgebraSort Action = new AlgebraSortImpl("Action");
AlgebraSort Channel = new AlgebraSortImpl("Channel");
AlgebraSort Data = new AlgebraSortImpl("Data");
Action.addConstructor("Emit", {"Chan_E", "Value_E"}, {Channel, Data});
Action.addConstructor("Receive", {"Chan_E", "Value_E"}, {Channel, Data});
Action.addConstructor("Tau");
```

C.2 Extended presentation, and Environment

In addition to the objects defined in the Algebra Presentation, there are specific objects that are introduced by the pNet construction, and by the semantic rules used to build the open transitions and their predicates. This includes, for a pNet *pnet*

- $Const(pnet)$: Constants from the pLTSs (controllers) transitions: these are new constant constructors, usually of sort Action, local to an instance of a controller. The pNet definition requires that all these constants are distinct from each other.
- $SVars(pnet)$: State variables of the controllers. Here also they are required to be distinct from those of other controllers.
- $IVars(pnet)$: Input variables of the controllers.
- $FVars(pnet, ot)$: Several kinds of "fresh" variables, created by application of rule Tr2, during the construction of each open transition *ot*: Action variables for the behaviour of holes and the resulting actions of transitions, variables created during the cloning of synchronisation vectors.

All variable sets above include their sort. To define the typing rules for expressions and the translation functions, for any pNet *pnet* and open transition *ot*, we define an extended presentation $\mathcal{P}_{pnet}$ and environment $\Gamma_{pnet,ot}$ that includes all of the objects above:

Definition C.2 *Given an algebra presentation* $\mathcal{P}_{pnet} =< Sorts, Constrs, Ops >$ *and a pNet pnet, we construct:*

- *An extended presentation* $\mathcal{P}_{pnet} < Sorts, Constrs \cup Const(pnet), Ops >$.
- *For a given open transition ot, an environment* $\Gamma_{pnet,ot} = SVars(pnet) \cup IVars(pnet) \cup FVars(pnet, ot)$.

D Well-formness and typing of expressions

The purpose of this section is to define static semantic notions that will guarantee that the translation to the SMT language will be correct, i.e. will not yield errors at runtime. This includes well-formness (all sorts, operators, variables are defined, and expressions respect the arity of operators), and typing rules.

Definition D.1 *Given a presentation* $\mathcal{P}$ *(possibly extended) and an environment* Γ*:*

- Γ *is well-formed if all sorts in* Γ *are defined in* $\mathcal{P}$
- *an expression is well-formed if all its operators are defined in* $\mathcal{P}$*, and used with the proper arity, and all its variables are defined in* Γ
- *an expression is well-typed if it can be typed by the typing rules in table 5*

The following judgment, and the typing rules in table 5 can be used to check both the wellformedness and well-typing of expressions in a pNet or in an open transition, given the corresponding $\mathcal{P}$ and Γ.

$\mathcal{P}, \Gamma \vdash M : A$	M is a well-formed term of type A in $\mathcal{P}, \Gamma$

Table 5: Type Rules for Open pNets

(Var x)

$$\frac{\Gamma \vdash x : A}{\mathcal{P}, \Gamma \vdash x : A}$$

(Binary operators, e.g.: $\wedge, \vee$ for Booleans, $+, -, \times, \div, \leq, \geq$ for integers, etc.)

$$\frac{\mathcal{P} \vdash BinOp :: ty1, ty1 \to ty2 \quad \Gamma \vdash x_1 : ty1 \quad \Gamma \vdash x_2 : ty1}{\mathcal{P}, \Gamma \vdash x_1 \ BinOp \ x_2 : ty2}$$

(Unary operators, e.g. $\neg$ for Booleans, - for integers)

$$\frac{\mathcal{P} \vdash UnOp :: ty1 \to ty2 \quad \Gamma \vdash x : ty1}{\mathcal{P}, \Gamma \vdash Unop \ x : ty2}$$

(Polymorphic EQ and NEQ)

$$\frac{\mathcal{P} \vdash A \quad \Gamma \vdash x_1 : A \quad \Gamma \vdash x_2 : A}{\mathcal{P}, \Gamma \vdash x_1 = x_2 : Bool} \qquad \frac{\mathcal{P} \vdash A \quad \Gamma \vdash x_1 : A \quad \Gamma \vdash x_2 : A}{\mathcal{P}, \Gamma \vdash x_1 \neq x_2 : Bool}$$

(Overloaded FUN operator)

$$\frac{\mathcal{P} \vdash \texttt{FUN} :: A_1, ..., A_n \to Action \quad \mathcal{P} \vdash A_1 \quad ... \quad \mathcal{P} \vdash A_n \quad \Gamma \vdash x_1 : A_1 \quad ... \quad \Gamma \vdash x_n : A_n}{\mathcal{P}, \Gamma \vdash \texttt{FUN}(x_1, ..., x_n) : Action}$$

Remark D.2 *These rules provide a simple type-checking algorithm: if all variables in an expression are known in* Γ*, then a bottom-up application of the rules will decide whether the expression is well-typed, and compute the type of each sub-expression.*

E Translation to SMTlib input language

The pNet elements, as defined above, can be full translated into SMT-LIB language ([20]), but there are a number of differences in the structure of the models languages, so the translation is not trivial.

In this section we define separately (as introduced in Section 5.3):

- The translation of the extended algebra presentation for one pNet (so it will be common for the study of all OTs of one use-case);
- and the translation of each open transition of the pNet, including its environment (list of variables), its predicate, and the encoding of the possible values of the state-variables of its source global state.

The following table summarizes the main elements of the translation.

(Presentation)	
	Sorts&Constructors $\hookrightarrow$ declare-datatypes
	Otheroperators $\hookrightarrow$ declare-fun
(Open Transition)	
	Γ $\hookrightarrow$ declare-const
	Pred $\hookrightarrow$ assert
	`pNet expression` $\hookrightarrow$ SMTLib expression
	assignments assert

In the following definitions, we use abstract functions corresponding to SMTlib constructs. In practice they can be implemented either as SMTlib scripting programs, or as calls to the Z3 java API.

E.1 Presentation Translation

We define here the translation of the algebra presentation, extended with the constant operators collected from the pNet. It produces the declaration of sorts (excepted Bool and Int) with their constructors and selectors as `declare-datatypes`, and the declaration of auxiliary operators as `declare-function` constructs.

For sorts, we must distinguish the case of mutually defined sorts (e.g. edges and vertices in a graph), that must be declared within a single `declare-datatypes` construct. For this we define a dependency order on sort, and build the strongly connected components of the graph of sorts:

Definition E.1 *Define the (strict) order "is-using" between sorts $S1$ $is-using$ $S2$ iff $S2$ occurs as the sort of one argument in the constructors of $S1$.*

Let Pres = <Sorts, Constrs, Ops> an extended presentation (i.e. including the constants from the pnet), - define MySorts = Sorts Bool, Int - compute the strongly connected components in the graph of MySorts with respect to the relation is-using - for each SCC in this graph, construct the definitions, using `declare-datatypes` for sorts, their constants and constructors, and `declare-function` for the other operators, using the following translation function:

One datatype declaration for each SCC

$$\frac{name = name(Sort) \qquad constrs = Constrs(name) \ \hookrightarrow constrs}{SCC \ \hookrightarrow \texttt{(declare-datatypes ()} \ name \ \texttt{(map} \ \hookrightarrow constrs\texttt{))}}$$

Constants:

$$\frac{arity(constr) = 0}{constr \ \hookrightarrow \texttt{name(}constr\texttt{)}}$$

Other constructors:

$$\frac{n = arity(constr) \neq 0 \qquad |selectors| = n \qquad selectors = BuildSels(constr)}{constr \ \hookrightarrow \texttt{(name(}constr\texttt{)} \ . \ selectors\texttt{)}}$$

Auxiliary operators:

$$\frac{\mathcal{P} \vdash op : sort_1, ..., sort_n : sort_0 \quad sortname_i = name(sort_i)}{op \ \hookrightarrow \texttt{(declare-fun name(}op\texttt{)} \ (sortname_1 ... sortname_n) \ sortname_0\texttt{)}}$$

Special case of FUN:

$$\frac{}{FUN \ \hookrightarrow \texttt{(map} \ \hookrightarrow BuildFunInstance(CollectFunTypes(pNet))\texttt{)}}$$

Where:

- the *BuildSels* function, for a constructor with n arguments, argument sorts $sort_i$, and selector names sel_i, builds the list $\{(sel_i, sort_i)\}_{i \in [1..n]}$.
- the *CollectFunTypes* function collects all possible instances of the overloaded FUN argument types found in the pNet, as computed by the typing rules; and *BuildFunInstance* use these argument types as suffixes to disambiguate the FUN operator name and build the corresponding `declare-fun` command.

Example E.2 *For the CCS presentation in example C.1, we would get:*

```
(declare-datatypes Data ())
(declare-datatypes Channel ())
(declare-datatypes ()
    ((Action (Emit (Chan_E Channel) (Value_E Data))
             (Receive (Chan_R Channel) (Value_R Data))
             Tau )))
```

Example E.3 *For our BIP use-case, we have built (see Fig.4):*

```
(declare-datatypes ()
    ((Action (fail) (resume) (timeout) (reset) (start) (tick) (ask)
  (FUN_Action_Bool (fst Action) (snd Bool))
  (Synchro (action Action)))))
```

E.2 Predicate translation

The second part of the translation function is called for each open transition. More precisely we need here:

Let Pres = <Sorts, Constrs, Ops> be the extended presentation and OT = <Leaves, Holes, Pred, Assign> an open transition.

- Environnement and correctness
  compute the environment $\Gamma = \Gamma_{pnet,OT}$ collecting all variables used in OT.
  check that all pLTS labels (action, guard, assignment) in the transition, the OT predicate, and the OT assignments are well-formed and well-typed.
  for each variable v in Γ, construct: define-const = TrPredicate (Γ, v).
- Assignments: for each start global state gs of the open transition, using $SVar(gs)$ get the state variables and their possible evaluations from all the state in gs. Then use Assigns(x) as the set of evaluations of state variable x.
- Predicate: Decompose the toplevel conjuncts of the predicate. For each conjunct P_i, construct: assert = TrPredicate (P_i).

In the following table, we denote $\hookrightarrow$ the main translation function, and $\longrightarrow$ the TrPredicate auxiliary function.

Variables:

$$\frac{\Gamma \vdash x : A}{x \hookrightarrow \texttt{(declare-const x A)}}$$

Open transition:

$$\frac{Pred \hookrightarrow lst_1 \quad Ass \hookrightarrow lst_2}{ot(Pred, Ass) \hookrightarrow lst_1 lst_2}$$

Assignment:

$$\frac{Ass = \bigwedge_{k \in K} (x_k, a_k) \quad x_k \in SVar(gs) \quad a_k = Assigns(x_k)}{Ass \hookrightarrow \texttt{(map} \hookrightarrow (x_k, a_k))}$$

$$\frac{a_k = e_1, ..., e_n}{(x_k, a_k) \hookrightarrow \texttt{(assert (or (= } x_k \texttt{ (map} \longrightarrow e_1\texttt{))...(= } x_k \texttt{ (map} \longrightarrow e_n\texttt{))))}}$$

Predicate:

$$\frac{Pred = \bigwedge_{k \in K} P_k}{Pred \hookrightarrow \texttt{map} \hookrightarrow P_k}$$

Predicate conjunct:

$$\frac{P_k = op(args)}{P_k \hookrightarrow \texttt{(assert (op map} \longrightarrow \texttt{args))}}$$

$$\frac{\Gamma \vdash x : A}{Var(x) \longrightarrow \texttt{x}}$$

$$\frac{arg = op'(args) \quad op' \in Constr \quad arity(op') = 0}{op' \longrightarrow name(op')}$$

$$\frac{arg = op'(args) \quad op' \in Constr \quad arity(op') \neq 0}{op' \longrightarrow \texttt{(}name(op') \quad (map \longrightarrow args)\texttt{)}}$$

$$\frac{arg = op'(args) \quad op' \in Ops}{op' \longrightarrow \texttt{(}name(op') \quad (map \longrightarrow args)\texttt{)}}$$

Properties: The rules in the sections above are meant to guarantee that no runtime errors would occur when running Z3 from our algorithm. More precisely, we ensure that:

- all datatypes, functions, variable used are declared,
- datatypes declared are well-founded,
- all expressions in declarations and assertions are well-typed.

F Full output for the FailureTimer open automaton

The following listing contains the final list of OTs generated for our use-case, as summarized in section 5.4 page 13. It is displayed in full and raw format, each OT containing in its premisses: the list of transitions of pLTSs and actions of holes, the predicate, the Post assignements. In the conclusion the transition labelled by a fresh "result" variable.

In Fig. 6 we display the same OTs in a more human friendly format (but manualy translated into latex).

```
   {s0---failure(true)-->s1, t0---start(true)-->t1},{---:hb_B:13:1-->},
   (failure(true)=failure(b1:sva_SV0:1:1))/\(start(true)=start(b2:sva_SV0:1:1))
   /\(:hb_B:13:1=fail(b0:sva_SV0:1:1))/\((b1:sva_SV0:1:1=b2:sva_SV0:1:1)
   /\(~(b1:sva_SV0:1:1\/b2:sva_SV0:1:1)\/b0:sva_SV0:1:1))
   /\(_fail_=:ra:1:1),{t := Max}
OT ------------------------------------------------------------------------------
        <s0_t0>-----:ra:1:1---><s1_t1>

   {s0---resume(false)-->s0, t0---resume(false)-->t0},{---:hb_B:16:2-->},
   (resume(false)=resume(b1:sva_SV1:1:2))
   /\(resume(false)=resume(b2:sva_SV1:1:2))
   /\(:hb_B:16:2=resume(b0:sva_SV1:1:2))/\((b1:sva_SV1:1:2=b2:sva_SV1:1:2)
   /\(~(b1:sva_SV1:1:2\/b2:sva_SV1:1:2)\/b0:sva_SV1:1:2)
   /\(_resume_=:ra:1:2),{}
OT ------------------------------------------------------------------------------
        <s0_t0>-----:ra:1:2---><s0_t0>

   {},{---:hb_B:118:6-->},(:hb_B:118:6=finish)/\(finish=:ra:1:6),{}
OT ------------------------------------------------------------------------------
        <s0_t0>-----:ra:1:6---><s0_t0>

   {s1---failure(false)-->s1, t1---start(false)-->t1},{---:hb_B:13:1-->},
   (failure(false)=failure(b1:sva_SV0:1:1))
   /\(start(false)=start(b2:sva_SV0:1:1))
   /\(:hb_B:13:1=fail(b0:sva_SV0:1:1))/\((b1:sva_SV0:1:1=b2:sva_SV0:1:1)
   /\(~(b1:sva_SV0:1:1\/b2:sva_SV0:1:1)\/b0:sva_SV0:1:1))
   /\(_fail_=:ra:1:1),{}
OT ------------------------------------------------------------------------------
        <s1_t1>-----:ra:1:1---><s1_t1>

   {s1---resume(true)-->s0, t1---resume(true)-->t0},{---:hb_B:16:2-->},
   (resume(true)=resume(b1:sva_SV1:1:2))/\(resume(true)=resume(b2:sva_SV1:1:2))
   /\(:hb_B:16:2=resume(b0:sva_SV1:1:2))/\((b1:sva_SV1:1:2=b2:sva_SV1:1:2)
   /\(~(b1:sva_SV1:1:2\/b2:sva_SV1:1:2)\/b0:sva_SV1:1:2))
   /\(_resume_=:ra:1:2),{}
OT ------------------------------------------------------------------------------
        <s1_t1>-----:ra:1:2---><s0_t0>

   {t1---tick-->t1},{},(t>0)/\(tick=:ra:1:3),{t := (t-1)}
OT ------------------------------------------------------------------------------
        <s1_t1>-----:ra:1:3---><s1_t1>

   {},{---:hb_B:118:6-->},(:hb_B:118:6=finish)/\(finish=:ra:1:6),{}
OT ------------------------------------------------------------------------------
        <s1_t1>-----:ra:1:6---><s1_t1>

   {s1---timeout-->s2, t1---timeout-->t0}, {},(t=0)/\(ask=:ra:1:4),{}
OT ------------------------------------------------------------------------------
        <s1_t1>-----:ra:1:4---><s2_t0>

   {s2---resume(false)-->s2, t0---resume(false)-->t0},{---:hb_B:16:2-->},
   (resume(false)=resume(b1:sva_SV1:1:2))
   /\(resume(false)=resume(b2:sva_SV1:1:2))
   /\(:hb_B:16:2=resume(b0:sva_SV1:1:2))/\((b1:sva_SV1:1:2=b2:sva_SV1:1:2)
   /\(~(b1:sva_SV1:1:2\/b2:sva_SV1:1:2)\/b0:sva_SV1:1:2))
   /\(_resume_=:ra:1:2),{}
OT ------------------------------------------------------------------------------
        <s2_t0>-----:ra:1:2---><s2_t0>

   {s2---reset-->s0},{},(reset=:ra:1:5),{}
OT ------------------------------------------------------------------------------
        <s2_t0>-----:ra:1:5---><s0_t0>

   {},{---:hb_B:118:6-->},(:hb_B:118:6=finish)/\(finish=:ra:1:6),{}
OT ------------------------------------------------------------------------------
        <s2_t0>-----:ra:1:6---><s2_t0>
```

F.1 Scaling up: several subsystems on the same bus

In our discussion 15, we mentionned an experiment done with a slightly larger example. We build a larger system by composing several subsystems. In the example here (Fig. 7), we use two Failure Monitoring modules. To compose them, a new controller is added for managing the reset function of the whole system. No matter which module emits an "ask" to call for a reset, the controller takes the request. Then all of the modules are synchronised on the "reset". So two new transitions are added to the Timer to ensure the module can reset at any state.

In this example, there are 327 satisfiable predicates for the total of 97920 candidate open transitions, computed in 105384 milliseconds (~1.45 minutes). The computation was carried out on an Intel Core i7 at 2.5GHz with 16GB of RAM.

Analysing this result, we found some extra states that did not exist when running only the subnets. Comparing this with the previous result on on a single subnet, we found that this was caused by an input mistake in a synchronisation vector.

After correction the algorithm produced 233280 OTs, whereof 281 are satisfiable. The resulting automaton has 17 reachable states. Of course we cannot display them here, and a model-checking tool would be required for their formal analysis.

$$ot_1 = \frac{\{s0 \xrightarrow{\texttt{fail(true)}} s1, t0 \xrightarrow{\texttt{start(true)}} t1\},\ \{\xrightarrow{hb_1}\},\ \texttt{fail(false)} = \texttt{fail}(\texttt{b}_1) \wedge \texttt{start(true)} = \texttt{start}(\texttt{b}_2) \wedge hb_1 = \texttt{fail}(\texttt{b}_0) \wedge v_1 = \underline{\texttt{fail}} \wedge \texttt{b}_1 = \texttt{b}_2 \wedge (\texttt{b}_1 \vee \texttt{b}_2) \Rightarrow \texttt{b}_0,\ \{t := Max\}}{\triangleleft s0, t0\triangleright \xrightarrow{v_1} \triangleleft s1, t1\triangleright}$$

$$ot_2 = \frac{\{s0 \xrightarrow{\texttt{resume(false)}} s0, t0 \xrightarrow{\texttt{resume(false)}} t0\},\ \{\xrightarrow{hb_2}\},\ \texttt{resume(false)} = \texttt{resume}(\texttt{b}_1) \wedge \texttt{resume(false)} = \texttt{resume}(\texttt{b}_2) \wedge hb_2 = \texttt{resume}(\texttt{b}_0) \wedge v_2 = \underline{\texttt{resume}} \wedge \texttt{b}_1 = \texttt{b}_2 \wedge (\texttt{b}_1 \vee \texttt{b}_2) \Rightarrow \texttt{b}_0,\ \{\}}{\triangleleft s0, t0\triangleright \xrightarrow{v_2} \triangleleft s0, t0\triangleright}$$

$$ot_3 = \frac{\{\},\ \{\xrightarrow{hb_3}\},\ hb_3 = \texttt{finish} \wedge v_3 = \texttt{finish},\ \{\}}{\triangleleft s0, t0\triangleright \xrightarrow{v_3} \triangleleft s0, t0\triangleright}$$

$$ot_4 = \frac{\{s1 \xrightarrow{\texttt{fail(false)}} s1, t1 \xrightarrow{\texttt{start(false)}} t1\},\ \{\xrightarrow{hb_4}\},\ \texttt{fail(false)} = \texttt{fail}(\texttt{b}_1) \wedge \texttt{start(false)} = \texttt{start}(\texttt{b}_2) \wedge hb_4 = \texttt{fail}(\texttt{b}_0) \wedge v_4 = \underline{\texttt{fail}} \wedge \texttt{b}_1 = \texttt{b}_2 \wedge (\texttt{b}_1 \vee \texttt{b}_2) \Rightarrow \texttt{b}_0,\ \{\}}{\triangleleft s1, t1\triangleright \xrightarrow{v_4} \triangleleft s1, t1\triangleright}$$

$$ot_5 = \frac{\{s1 \xrightarrow{\texttt{resume(true)}} s0, t1 \xrightarrow{\texttt{resume(true)}} t0\},\ \{\xrightarrow{hb_5}\},\ \texttt{resume(true)} = \texttt{resume}(\texttt{b}_1) \wedge \texttt{resume(true)} = \texttt{resume}(\texttt{b}_2) \wedge hb_2 = \texttt{resume}(\texttt{b}_0) \wedge v_5 = \underline{\texttt{resume}} \wedge \texttt{b}_1 = \texttt{b}_2 \wedge (\texttt{b}_1 \vee \texttt{b}_2) \Rightarrow \texttt{b}_0,\ \{\}}{\triangleleft s1, t1\triangleright \xrightarrow{v_5} \triangleleft s0, t0\triangleright}$$

$$ot_6 = \frac{\{t1 \xrightarrow{\texttt{tick}} t1\},\ \{\},\ t \neq 0 \wedge v_6 = \texttt{tick},\ \{t := t - 1\}}{\triangleleft s1, t1\triangleright \xrightarrow{v_6} \triangleleft s1, t0\triangleright}$$

$$ot_7 = \frac{\{\},\ \{\xrightarrow{hb_7}\},\ hb_7 = \texttt{finish} \wedge v_7 = \texttt{finish},\ \{\}}{\triangleleft s1, t1\triangleright \xrightarrow{v_7} \triangleleft s1, t1\triangleright}$$

$$ot_8 = \frac{\{s1 \xrightarrow{\texttt{timeout}} s2, t1 \xrightarrow{\texttt{timeout}} t0\},\ \{\},\ t = 0 \wedge v_8 = ask,\ \{\}}{\triangleleft s1, t1\triangleright \xrightarrow{v_8} \triangleleft s2, t0\triangleright}$$

$$ot_9 = \frac{\{s2 \xrightarrow{\texttt{resume(false)}} s2, t0 \xrightarrow{\texttt{resume(false)}} t0\},\ \{\xrightarrow{hb_9}\},\ \texttt{resume(false)} = \texttt{resume}(\texttt{b}_1) \wedge \texttt{resume(false)} = \texttt{resume}(\texttt{b}_2) \wedge hb_9 = \texttt{resume}(\texttt{b}_0) \wedge v_9 = \underline{\texttt{resume}} \wedge \texttt{b}_1 = \texttt{b}_2 \wedge (\texttt{b}_1 \vee \texttt{b}_2) \Rightarrow \texttt{b}_0,\ \{\}}{\triangleleft s2, t0\triangleright \xrightarrow{v_9} \triangleleft s2, t0\triangleright}$$

$$ot_{10} = \frac{\{s2 \xrightarrow{\texttt{reset}} s0\},\ \{\},\ v_{10} = \texttt{reset},\ \{\}}{\triangleleft s2, t0\triangleright \xrightarrow{v_{10}} \triangleleft s0, t0\triangleright}$$

$$ot_{11} = \frac{\{\},\ \{hb_{11}\},\ hb_{11} = \texttt{finish} \wedge v_{11} = \texttt{finish},\ \{\}}{\triangleleft s2, t0\triangleright \xrightarrow{v_{11}} \triangleleft s2, t0\triangleright}$$

Figure 6: The open transitions of the open automaton

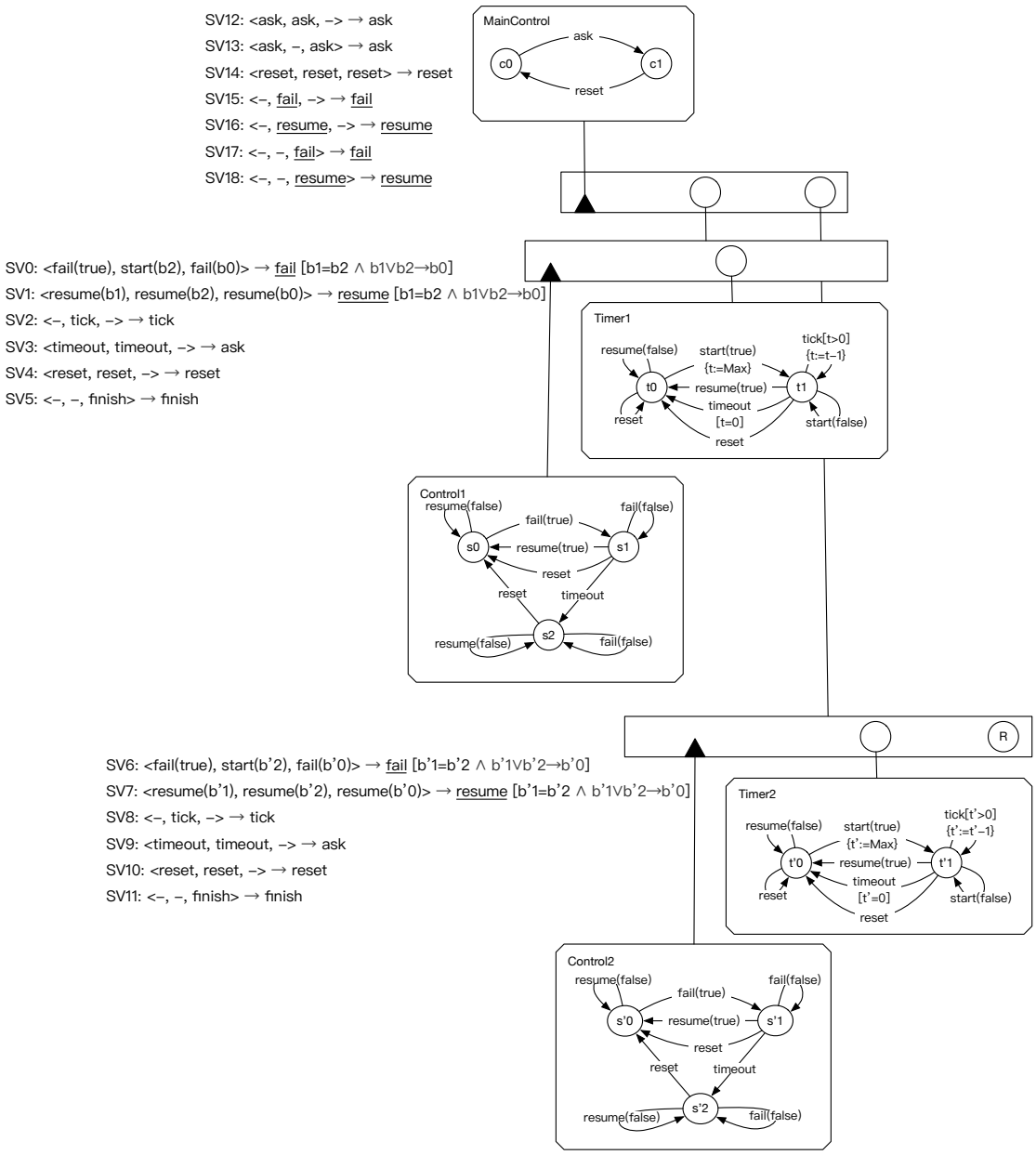
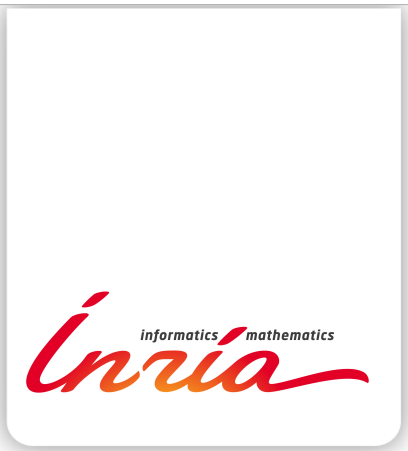


Figure 7: pNet encoding of the composition of two Failure Monitoring modules



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399